Typesetting in Lua using LuaTEX

Introduction

Sometimes you hear folks complain about the TEX input language, i.e. the backslashed commands that determine your output. Of course, when alternatives are being discussed every one has a favourite programming language. In practice coding a document in each of them triggers similar sentiments with regards to coding as TEX itself does.

So, just for fun, I added a couple of commands to ConTEXt MkIV that permit coding a document in Lua. In retrospect it has been surprisingly easy to implement a feature like this using metatables. Of course it's a bit slower than using TEX as input language but sometimes the Lua interface is more readable given the problem at hand.

After a while I decided to use that interface in non-critical core ConTEXt code and in styles (modules) and solutions for projects. Using the Lua approach is sometimes more convenient, especially if the code mostly manipulates data. For instance, if you process xml files of database output you can use the interface that is available at the TEX end, or you can use Lua code to do the work, or you can use a combination. So, from now on, in ConTEXt you can code your style and document source in (a mixture of) TEX, xml, MetaPost and in Lua.

In this article I will introduce typesetting in Lua, but as we rely on ConTEXt it is unavoidable that some regular ConTEXt code shows up. The fact that you can ignore backslashes does not mean that you can do without knowledge of the underlying system. I assume the user is somewhat familiar with this macro package.

Some basics

To start with, I assume that you have either the so called ConTEXt minimals installed or TEXLive. You only need LuaTEX and can forget about installing pdfTEX or XETEX, which saves you some megabytes and hassle. Now, from the user's perspective a ConTEXt run goes like:

```
context yourfile
```

and by default a file with suffix `tex` will be processed. There are however a few other options:

```
context yourfile.xml
context yourfile.rlx --forcexml
context yourfile.lua
context yourfile.pqr --forcelua
context yourfile.cld
context yourfile.xyz --forcecld
```

When processing a Lua file the given file is loaded and just processed. This option will seldom be used as it is way more efficient to let `mtxrun` process that file. However, the last two variants are what we will discuss here. The suffix `cld` is a shortcut for ConTEXt Lua Document.

A simple `cld` file looks like this:

```
context.starttext()
context.chapter("Hello There!")
context.stoptext()
```

So yes, you need to know the ConTEXt commands in order to use this mechanism. In spite of what you might expect, the codebase involved in this interface is not that large. If you know ConTEXt, and if you know how to call commands, you basically can use this Lua method.
The examples that I will give are either (sort of) standalone, that is, they are dealt with from Lua, or they are run within this document. Therefore you will see two patterns. If you want to make your own documentation, then you can use this variant:

```
\startbuffer
context("See this!")
\stopbuffer

\typebuffer \ctxluabuffer
```

I use anonymous buffers here but you can also use named ones. The other variant is:

```
\startluacode
context("See this!")
\stopluacode
```

This will process the code directly. Of course we could have encoded this document completely in Lua but that is not much fun for a manual.

The main command

There are a few rules that you need to be aware of. First of all no syntax checking is done. Second you need to know what the given commands expects in terms of arguments. Third, the type of your arguments matters:

```
nothing : just the command, no arguments
string  : an argument with curly braces
array   : a list between square backets (sometimes optional)
hash    : an assignment list between square brackets
boolean : when true a newline is inserted
        : when false, omit braces for the next argument
```

In the code above you have seen examples of this but here are some more:

```
context.chapter("Some title")
context.chapter({ "first" }, "Some title")
context.startchapter({ title = "Some title", label = "first" })
```

This blob of code is equivalent to:

```
\chapter{Some title}
\chapter[first]{Some title}
\startchapter[title={Some title},label=first]
```

You can simplify the third line of the Lua code to:

```
context.startchapter { title = "Some title", label = "first" }
```

In case you wonder what the distinction is between square brackets and curly braces: the first category of arguments concerns settings or lists of options or names of instances while the second category normally concerns some text to be typeset.

Strings are interpreted as TEX input, so:

```
context.mathematics("\\sqrt{2^3}")
```

or, if you don't want to escape:

```
context.mathematics([[\sqrt{2^3}]])
```

is okay. As TEX math is a language in its own and a de-facto standard way of inputting math this is quite natural, even at the Lua end.

Spaces and Lines

In a regular TEX file, spaces and newline characters are collapsed into one space. At the Lua end the same happens. Compare the following examples. First we omit spaces:

```
context("left")
context("middle")
context("right")
```

leftmiddleright

Next we add spaces:

```
context("left")
context(" middle ")
context("right")
```

left middle right

We can also add more spaces:

```
context("left ")
context(" middle ")
context(" right")
```

left middle right

In principle all content becomes a stream and after that the TEX parser will do its normal work: collapse spaces unless configured to do otherwise. Now take the following code:

```
context("before")
context("word 1")
context("word 2")
```

```
context("word 3")
context("after")
```

beforeword 1word 2word 3after

Here we get no spaces between the words at all, which is what we expect. So, how do we get lines (or paragraphs)?

```
context("before")
context.startlines()
context("line 1")
context("line 2")
context("line 3")
context.stoplines()
context("after")
```

before

line 1line 2line 3

after

This does not work out well, as again there are no lines seen at the TEX end. Newline tokens are injected by passing `true` to the `context` command:

```
context("before")
context.startlines()
context("line 1") context(true)
context("line 2") context(true)
context("line 3") context(true)
context.stoplines()
context("after")
```

before

line 1
line 2
line 3

after

Don't confuse this with:

```
context("before") context.par()
context("line 1") context.par()
context("line 2") context.par()
context("line 3") context.par()
context("after")  context.par()
```

before
line 1
line 2
line 3
after

There we use the regular \par command to finish the current paragraph and normally you will use that method. In that case, when set, whitespace will be added between paragraphs.

Direct output

The ConTEXt user interface is rather consistent and the use of special input syntaxes is discouraged. Therefore, the Lua interface using tables and strings works quite well. However, imagine that you need to support some weird macro (or a primitive) that does not expect its argument between curly braces or brackets. The way out is to precede an argument by another one with the value false. We call this the direct interface. This is demonstrated in the following example.

```
\unexpanded\def\bla#1{[#1]}

\startluacode
context.bla(false,"***")
context.par()
context.bla("***")
\stopluacode
```

This results in:

[*]**
[***]

Here, the first call results in three * being passed, and #1 picks up the first token. The second call to bla gets {***} passed so here #1 gets the triplet. In practice you will seldom need the direct interface.

In ConTEXt for historical reasons, combinations have the following syntax:

```
\startcombination % optional specification, like [2*3]
  {\framed{content one}} {caption one}
  {\framed{content two}} {caption two}
\stopcombination
```

You can also say:

```
\startcombination
  \combination {\framed{content one}} {caption one}
  \combination {\framed{content two}} {caption two}
\stopcombination
```

When coded in Lua, we can feed the first variant as follows:

```
context.startcombination()
  context.direct("one","two")
  context.direct("one","two")
context.stopcombination()
```

To give you an idea what this looks like, we render it:

one one
two two

So, the `direct` function is basically a no-op and results in nothing by itself. Only arguments are passed. Equivalent, but a bit more ugly looking, is:

```
context.startcombination()
  context(false,"one","two")
  context(false,"one","two")
context.stopcombination()
```

Catcodes

If you are familiar with TEX's inner working, you will know that characters can have special meanings. This meaning is determined by the characters catcode.

```
context("$x=1$")
```

This gives: $x = 1$ because the dollar tokens trigger inline math mode. If you think that this is annoying, you can do the following:

```
context.pushcatcodes("text")
context("$x=1$")
context.popcatcodes()
```

Now we get: $x=1$. There are several catcode regimes of which only a few make sense in the perspective of the cld interface.

`ctx, ctxcatcodes, context`	the normal ConTEXt catcode regime
`prt, prtcatcodes, protect`	the ConTEXt protected regime, used for modules
`tex, texcatcodes, plain`	the traditional (plain) TEX regime
`txt, txtcatcodes, text`	the ConTEXt regime but with less special characters
`vrb, vrbcatcodes, verbatim`	a regime specially meant for verbatim
`xml, xmlcatcodes`	a regime specially meant for xml processing

In the second case you can still get math:

```
context.pushcatcodes("text")
context.mathematics("x=1")
context.popcatcodes()
```

When entering a lot of math you can also consider this:

```
context.startimath()
context("x")
context("=")
context("1")
context.stopimath()
```

Module writers of course can use `unprotect` and `protect` as they do at the TEX end. As we've seen, a function call to `context` acts like a print, as in:

```
context("test ")
context.bold("me")
context(" first")
```

test **me** first

When more than one argument is given, the first argument is considered a format conforming the `string.format` function.

```
context.startimath()
context("%s = %0.5f",utf.char(0x03C0),math.pi)
context.stopimath()
```

$\pi = 3.14159$

This means that when you say:

```
context(a,b,c,d,e,f)
```

the variables `b` till `f` are passed to the format and when the format does not call for them, they will not end up in your output.

```
context("%s %s %s",1,2,3)
context(1,2,3)
```

The first line results in the three numbers being typeset, but in the second case only the number 1 is typeset.

Why we need functions

In a previous section we introduced functions as arguments. At first sight this feature looks strange but you need to keep in mind that a call to a `context` function has no direct consequences. It generates TEX code that is executed after the current Lua chunk ends and control is passed back to TEX. Take the following code:

```
context.framed( {
    frame = "on",
    offset = "5mm",
    align = "middle"
  },
  context.input("knuth")
)
```

We call the function `framed` but before the function body is executed, the arguments get evaluated. This means that `input` gets processed before `framed` gets done. As a result there is no second argument to `framed` and no content gets passed: an error is reported. This is why we need the indirect call:

```
context.framed( {
    frame = "on",
    align = "middle"
  },
  function() context.input("knuth") end
)
```

This way we get what we want:

> Thus, I came to the conclusion that the designer of a new system must not only be the implementer and first large–scale user; the designer should also write the first user manual. The separation of any of these four components would have hurt TEX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important. But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.

The function is delayed till the `framed` command is executed. If your applications use such calls a lot, you can of course encapsulate this ugliness:

```
mycommands = mycommands or { }

function mycommands.framed_input(filename)
  context.framed( {
    frame = "on",
    align = "middle"
  },
  function() context.input(filename) end
end

mycommands.framed_input("knuth")
```

Of course you can nest function calls:

```
context.placefigure(
  "caption",
  function()
    context.framed( {
      frame = "on",
      align = "middle"
    },
      function() context.input("knuth") end
    )
  end
)
```

Or you can use a more indirect method:

```
function text()
  context.framed( {
      frame = "on",
      align = "middle"
    },
    function() context.input("knuth") end
  )
end

context.placefigure(
  "none",
  function() text() end
)
```

You can develop your own style and libraries just like you do with regular Lua code.

How we can avoid them

As many nested functions can obscure the code rather quickly, there is an alternative. In the following examples we use `test`:

```
\def\test#1{[#1]}

context.test("test 1",context(" test 2a "),"test 3")
```

This gives: test 2a [test 1]test 3. As you can see, the second argument is executed before the encapsulating call to `test`. So, we should have packed it into a function but here is an alternative:

```
context.test("test 1",context.delayed(" test 2a "),"test 3")
```

Now we get: [test 1] test 2a test 3. We can also delay functions themselves, look at this:

```
context.test("test 1",context.delayed.test(" test 2b "),"test 3")
```

The result is: [test 1][test 2b]test 3. This feature also conveniently permits the use of temporary variables, as in:

```
local f = context.delayed.test(" test 2c ")
context("before",f,"after")
```

Of course you can limit the amount of keystrokes even more by creating a shortcut:

```
local delayed = context.delayed

context.test("test 1",delayed.test(" test 2 "),"test 3")
context.test("test 4",delayed.test(" test 5 "),"test 6")
```

So, if you want you can produce rather readable code and readability of code is one of the reasons why Lua was chosen in the first place.
There is also another mechanism available. In the next example the second argument is actually a string.

```
local nested = context.nested

context.test("test 8",nested.test("test 9"),"test 10")
```

There is a pitfall here: a nested context command needs to be flushed explicitly, so in the case of:

```
context.nested.test("test 9")
```

a string is created but nothing ends up at the TEX end. Flushing is up to you. Beware: nested only works with the regular ConTEXt catcode regime.

Trial typesetting

Some typesetting mechanisms demand a preroll. For instance, when determining the most optimal way to analyse and therefore typeset a table, it is necessary to

typeset the content of cells first. Inside ConTEXt there is a state tagged 'trial typesetting' which signals other mechanisms that for instance counters should not be incremented more than once.
Normally you don't need to worry about these issues, but when writing the code that implements the Lua interface to ConTEXt, it definitely had to be taken into account as we either or not can free cached (nested) functions.
You can influence this caching to some extend. If you say

```
function()
  context("whatever")
end
```

the function will be removed from the cache when ConTEXt is not in the trial typesetting state. You can prevent *any* removal of a function by returning `true`, as in:

```
function()
  context("whatever")
  return true
end
```

Whenever you run into a situation that you don't get the outcome that you expect, you can consider returning `true`. However, keep in mind that it will take more memory, something that only matters on big runs. You can force flushing the whole cache by:

```
context.restart()
```

An example of an occasion where you need to keep the function available is in repeated content, for instance in headers and footers.

```
context.setupheadertexts {
  function()
    context.pagenumber()
    return true
  end
}
```

Of course it is not needed when you use the following method:

```
context.pagenumber("pagenumber")
```

Because here ConTEXt itself deals with the content driven by the keyword `pagenumber`.

Variables

Normally it makes most sense to use the English version of ConTEXt. The advantage is that you can use English keywords, as in:

```
context.framed( {
    frame = "on",
  },
  "some text"
)
```

If you use the Dutch interface it looks like this:

```
context.omlijnd( {
    kader = "aan",
  },
  "wat tekst"
)
```

A rather neutral way is:

```
context.framed( {
    frame = interfaces.variables.on,
  },
  "some text"
)
```

But as said, normally you will use the English user interface so you can forget about these matters. However, in the ConTEXt core code you will often see the variables being used this way because there we need to support all user interfaces.

Modes

Context carries a concept of modes. You can use modes to create conditional sections in your style (and/or content). You can control modes in your styles or you can set them at the command line or in job control files. When a mode test has to be done at processing time, then you need constructs like the following:

```
context.doifmodeelse( "screen",
  function()
      ... -- mode == screen
  end,
  function()
      ... -- mode ~= screen
  end
)
```

However, often a mode does not change during a run, and then we can use the following method:

```
if tex.modes["screen"] then
  ...
else
  ...
end
```

Watch how the `modes` table lives in the `tex` namespace. We also have `systemmodes`. At the TEX end these are mode names preceded by a `*`, so the following code is similar:

```
if tex.modes["*mymode"] then
  -- this is the same
elseif tex.systemmodes["mymode"] then
  -- test as this
else
  -- but not this
end
```

Inside ConTEXt we also have so called constants, and again these can be consulted at the Lua end:

```
if tex.constants["someconstant"] then
  ...
else
  ...
end
```

But you will hardly need these and, as they are often not public, their meaning can change, unless of course they *are* documented as public.

Token lists

There is normally no need to mess around with nodes and tokens at the Lua end yourself. However, if you do, then you might want to flush them as well. Say that at the TEX end we have said:

```
\toks0 = {Don't get \inframed{framed}!}
```

Then at the Lua end you can say:

```
context(tex.toks[0])
```

and get: Don't get framed! In fact, token registers are exposed as strings so here, register zero has type `string` and is treated as such.

```
context("< %s >",tex.toks[0])
```

This gives: < Don't get framed! >. But beware, if you go the reverse way, you don't get what you might expect:

```
tex.toks[0] = [[\framed{oeps}]]
```

If we now say `\the\toks0` we will get `Don't get framed!` as all tokens are considered to be letters.

Node lists

If you're not deep into TEX you will never feel the need to manipulate nodelists yourself, but you might want to flush boxes. As an example we put something in box zero (one of the scratch boxes).

```
\setbox0 = \hbox{Don't get \inframed{framed}!}
```

At the TEX end you can flush this box (`\box0`) or take a copy (`\copy0`). At the Lua end you would do:

```
context.copy()
context.direct(0)
```

or:

```
context.copy(false,0)
```

but this works as well:

```
context(node.copy_list(tex.box[0]))
```

So we get: Don't get framed! If you do:

```
context(tex.box[0])
```

you also need to make sure that the box is freed but let's not go into those details now.

Styles

Say that you want to typeset a word in a bold font. You can do that this way:

```
context("This is ")
context.bold("important")
context("!")
```

Now imagine that you want this important word to be in red too. As we have a nested command, we end up with a nested call:

```
context("This is ")
context.bold(function() context.color( { "red" }, "important") end)
context("!")
```

or

```
context("This is ")
context.bold(context.delayed.color( { "red" }, "important"))
context("!")
```

In that case it's good to know that there is a command that combines both features:

```
context("This is ")
context.style( { style = "bold", color = "red" }, "important")
context("!")
```

But that is still not convenient when we have to do that often. So, you can wrap the style switch in a function.

```
local function mycommands.important(str)
    context.style( { style = "bold", color = "red" }, str )
end

context("This is ")
mycommands.important( "important")
context(", and ")
mycommands.important( "this")
context(" too !")
```

Or you can setup a named style:

```
context.setupstyle( { "important" }, { style = "bold", color = "red" } )

context("This is ")
context.style( { "important" }, "important")
context(", and ")
context.style( { "important" }, "this")
context(" too !")
```

Or even define one:

```
context.definestyle( { "important" }, { style = "bold", color = "red" } )

context("This is ")
context.important("important")
context(", and ")
context.important("this")
context(" too !")
```

This last solution is especially handy for more complex cases:

```
context.definestyle( { "important" }, { style = "bold", color = "red" } )

context("This is ")
context.startimportant()
context.inframed("important")
context.stopimportant()
context(", and ")
context.important("this")
context(" too !")
```

This is **important**, and **this** too !

A complete example

One day my 6 year old niece Lorien was at the office and wanted to know what I was doing. As I knew she was practicing calculus at school I wrote a quick and dirty script to generate sheets with exercises. The most impressive part was that the answers were included. It was a rather braindead bit of Lua, written in a few minutes, but the weeks after I ended up running it a few more times, for her and her friends, every time a bit more difficult and also using different calculus. It was that script that made me decide to extend the basic cld manual into this more extensive document. We generate three columns of exercises. Each exercise is a row in a table. The last argument to the function determines if answers are shown.

```
local random = math.random

local function ForLorien(n,maxa,maxb,answers)
  context.startcolumns { n = 3 }
  context.starttabulate { "|r|c|r|c|r|" }
  for i=1,n do
    local sign = random(0,1) > 0.5
    local a, b = random(1,maxa or 99), random(1,max or maxb or 99)
    if b > a and not sign then a, b = b, a end
    context.NC()
    context(a)
```

```
    context.NC()
    context.mathematics(sign and "+" or "-")
    context.NC()
    context(b)
    context.NC()
    context("=")
    context.NC()
    context(answers and (sign and a+b or a-b))
    context.NC()
    context.NR()
  end
  context.stoptabulate()
  context.stopcolumns()
  context.page()
end
```

This is a typical example of where it's more convenient to write the code in Lua that in TEX's macro language. As a consequence setting up the page also happens in Lua:

```
context.setupbodyfont {
  "palatino",
  "14pt"
}
context.setuplayout {
  backspace = "2cm",
  topspace  = "2cm",
  header    = "1cm",
  footer    = "0cm",
  height    = "middle",
  width     = "middle",
}
```

At this point, we need to generate the document. There is a pitfall here: we need to use the same random number for the exercises and the answers, so we freeze and defrost it. Functions in the commands namespace implement functionality that is used at the TEX end but better can be done in Lua than in TEX macro code. Of course these functions can also be used at the Lua end.

```
context.starttext()
  local n = 120
  commands.freezerandomseed()
  ForLorien(n,10,10)
  ForLorien(n,20,20)
  ForLorien(n,30,30)
  ForLorien(n,40,40)
  ForLorien(n,50,50)
  commands.defrostrandomseed()
  ForLorien(n,10,10,true)
  ForLorien(n,20,20,true)
  ForLorien(n,30,30,true)
  ForLorien(n,40,40,true)
  ForLorien(n,50,50,true)
context.stoptext()
```

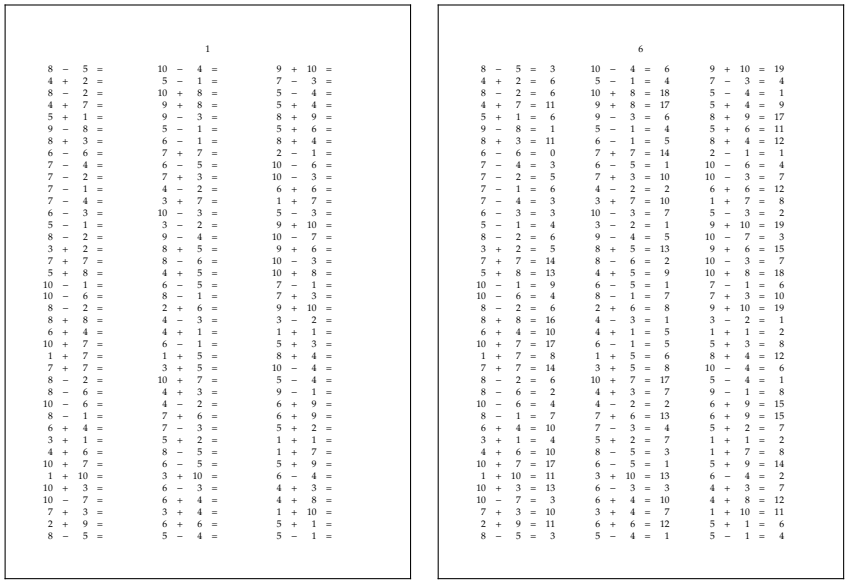


Figure 1 Lorien's challenge.

A few pages of the result are shown in figure 1. In the ConTEXt distribution more advanced version can be found in `s-edu-01.cld` as I was also asked to generate multiplication and table exercises. I also had to make sure that there were no duplicates on a page as she complained that was not good. There a set of sheets is generated with:

```
moduledata.educational.calculus.generate {
  name     = "Bram Otten",
  fontsize = "12pt",
  columns  = 2,
  run      = {
    { method = "bin_add_and_subtract", maxa =   8, maxb =   8 },
    { method = "bin_add_and_subtract", maxa =  16, maxb =  16 },
    { method = "bin_add_and_subtract", maxa =  32, maxb =  32 },
    { method = "bin_add_and_subtract", maxa =  64, maxb =  64 },
    { method = "bin_add_and_subtract", maxa = 128, maxb = 128 },
  },
}
```

Graphics

If you are familiar with ConTEXt, which by now probably is the case, you will have noticed that it integrates the MetaPost graphic subsystem. Drawing a graphic is not that complex:

```
context.startMPcode()
context [[
  draw
    fullcircle scaled 1cm
    withpen pencircle scaled 1mm
    withcolor .5white
```

```
    dashed dashpattern (on 2mm off 2mm) ;
 ]]
context.stopMPcode()
```

We get a gray dashed circle rendered with an one millimeter thick line:

So, we just use the regular commands and pass the drawing code as strings. Although MetaPost is a rather normal language and therefore offers loops and conditions and the lot, you might want to use Lua for anything else than the drawing commands. Of course this is much less efficient, but it could be that you don't care about speed. The next example demonstrates the interface for building graphics piecewise.

```
context.resetMPdrawing()

context.startMPdrawing()
context([[fill fullcircle scaled 5cm withcolor (0,0,.5) ;]])
context.stopMPdrawing()

context.MPdrawing("pickup pencircle scaled .5mm ;")
context.MPdrawing("drawoptions(withcolor white) ;")

for i=0,50,5 do
  context.startMPdrawing()
  context("draw fullcircle scaled %smm ;",i)
  context.stopMPdrawing()
end

for i=0,50,5 do
  context.MPdrawing("draw fullsquare scaled " .. i .. "mm ;")
end

context.MPdrawingdonetrue()

context.getMPdrawing()
```

This gives:

I the first loop we can use the format options associated with the simple `context` call. This will not work in the second case. Even worse, passing more than one argument will definitely give a faulty graphic definition. This is why we have a special interface for MetaFun. The code above can also be written as:

```
local metafun = context.metafun
```

```
metafun.start()

metafun("fill fullcircle scaled 5cm withcolor %s ;",
    metafun.color("darkblue"))

metafun("pickup pencircle scaled .5mm ;")
metafun("drawoptions(withcolor white) ;")

for i=0,50,5 do
  metafun("draw fullcircle scaled %smm ;",i)
end

for i=0,50,5 do
  metafun("draw fullsquare scaled %smm ;",i)
end

metafun.stop()
```

Watch the call to color, this will pass definitions at the TEX end to MetaPost. Of course you really need to ask yourself "Do I want to use MetaPost this way?" Using Lua loops instead of MetaPost ones makes much more sense in the following case:

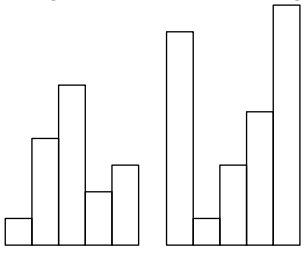
```
local metafun = context.metafun

function metafun.barchart(t)
  metafun.start()
  local t = t.data
  for i=1,#t do
    metafun("draw unitsquare xyscaled(%s,%s) shifted (%s,0);",
      10, t[i]*10, i*10)
  end
  metafun.stop()
end

local one = { 1, 4, 6, 2, 3, }
local two = { 8, 1, 3, 5, 9, }

context.startcombination()
  context.combination(metafun.delayed.barchart { data = one }, "one")
  context.combination(metafun.delayed.barchart { data = two }, "two")
context.stopcombination()
```

We get two barcharts alongside:



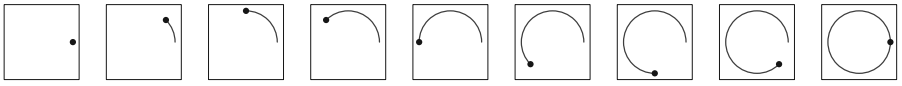
one two

```
local template = [[
  path p, q ; color c[] ;
  c1 := \MPcolor{darkblue} ;
  c2 := \MPcolor{darkred} ;
  p := fullcircle scaled 50 ;
```

```
  l := length p ;
  n := %s ;
  q := subpath (0,%s/n*l) of p ;
  draw q withcolor c2 withpen pencircle scaled 1 ;
  fill fullcircle scaled 5 shifted point length q of q withcolor c1 ;
  setbounds currentpicture to unitsquare shifted (-0.5,-0.5) scaled 60 ;
  draw boundingbox currentpicture withcolor c1 ;
  currentpicture := currentpicture xsized(1cm) ;
]]

local function steps(n)
  for i=0,n do
    context.metafun.start()
      context.metafun(template,n,i)
    context.metafun.stop()
    if i < n then
        context.quad()
    end
  end
end

context.hbox(function() steps(8) end)
```



To some extent we fool ourselves with this kind of Luafication of MetaPost code. Of course we can make a nice MetaPost library and put the code in a macro instead. In that sense, doing this in ConTEXt directly often gives better and more efficient code. Of course you can use all relevant commands in the Lua interface, like:

```
context.startMPpage()
  context("draw origin")
  for i=0,100,10 do
    context("..{down}(%d,0)",i)
  end
  context(" withcolor \\MPcolor{darkred} ;")
context.stopMPpage()
```

to get a graphic that has its own page. Don't use the `metafun` namespace here, as it will not work here. This drawing looks like:

Hans Hagen